

# COMP 617 RAP Seminar, Fall 2006

Presenter: Walid Taha

Scribe: Roumen Kaiabachev

Date: 26 Sep 2006

## 1 Simply Typed Lambda Calculus

We add types to the lambda calculus to classify lambda terms:

Expressions	$e ::= x \mid \lambda x.e \mid e e \mid T \mid F$	
Types	$\Pi \ni t ::= \text{Bool} \mid t \rightarrow t$	
Environment	$\Gamma ::= [] \mid \Gamma, x \mapsto t$	(1)
	$\Gamma ::= [] \mid (x, t) :: \Gamma$	(2)
	$\Gamma ::= X \rightarrow \Pi$	(3)
	$\Gamma ::= [] \mid \Gamma, t$	(4)

The choice of an environment definition  $\Gamma$  is important. For example, (4) does not use variables, so it is suitable for classifying de Bruijn terms.

We define a *typing relation* for the lambda calculus  $t : T$  by a set of inference rules. For variables, we have:

$$\frac{\Gamma \equiv \Gamma', x \mapsto t \text{ (1)}}{\Gamma \vdash x : t} \quad \text{OR} \quad \frac{\Gamma(x) = t \text{ (3)}}{\Gamma \vdash x : t}$$

The rule which uses option (1) for an environment is generally accepted. Alternatively, for unnamed variables using de Bruijn we have:

$$\frac{\Gamma \equiv \Gamma', t \text{ (4)}}{\Gamma \vdash \#0 : t} \quad \frac{\Gamma \equiv \Gamma', t' \quad \Gamma' \vdash \#n : t}{\Gamma \vdash \#n + 1 : t}$$

The de Bruijn notation avoids ambiguities in  $\Gamma$  because the binding occurrences of variables are uniquely identified by their insertion order in  $\Gamma$ . The rules say that there is no typing derivation if the environment is empty. In the second rule, we are not interested in the type of the 0th variable which type we drop from  $\Gamma$ .

The typing relation for named-variable lambda terms is given by the inference rule:

$$\frac{\Gamma, x \mapsto t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \rightarrow t_2}$$

The rule adds a binding for  $x$  in the environment and types the body of the lambda given that extension.

## 2 Types and Constructive Logic

Recall the definition of types and add times and union to possible types:

$$t ::= Bool \mid \dots \mid t \times t \mid t + t$$

Consider the BNF of constructive logic:

$$P ::= x \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid (\forall x.P \mid \exists x.P)$$

Let's look at the sequent calculus formulation where  $\Gamma$  denotes a sequence of propositions which is also a set of assumptions:

$$\frac{P \in \Gamma}{\Gamma \vdash P} \quad \frac{\Gamma, P_1 \vdash P_2}{\Gamma \vdash P_1 \Rightarrow P_2}$$

$$\frac{\Gamma \vdash P_1 \Rightarrow P_2 \quad \Gamma \vdash P_1}{\Gamma \vdash P_2} \text{ (modus ponens)} \quad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2}$$

### 2.1 Tree Linearization

In the types derivation for the lambda calculus, the terms capture the form of the tree. By just looking at the lambda term on the bottom, we can tell exactly what rules would be used to derive the proof. Is it the same case for our formulation above? Consider the first rule and the modus ponens rule above. By solely looking at the judgements, one cannot tell which rule was used during the derivation. We want a way to linearize a tree, while still capturing the individual rules used throughout the proof.

We rewrite each rule by labeling each proposition as follows:

$$\frac{l : P \in \Gamma}{\Gamma \vdash l : P} \quad \frac{\Gamma, l : P_1 \vdash e : P_2}{\Gamma \vdash (l, e) : P_1 \Rightarrow P_2}$$

$$\frac{\Gamma \vdash e_1 : P_1 \Rightarrow P_2 \quad \Gamma \vdash e_2 : P_1}{\Gamma \vdash e_1 @ e_2 : P_2} \text{ (modus ponens)} \quad \frac{\Gamma \vdash e_1 : P_1 \quad \Gamma \vdash e_2 : P_2}{\Gamma \vdash (e_1, e_2) : P_1 \wedge P_2}$$

Now we can just show the judgement to people and they would be able to know how we got the result.

Note that these rules look very similar to the typing rules for the lambda calculus. Specifically, the first rule looks like the variable rule, the next rule looks like the abstraction rule, and the third rule looks like the application rule. This leads to the *Curry-Howard isomorphism*, which says that the type of an expression computed is analogous to a logical theorem, and that the typing derivation tree to compute that value is analogous to a proof of that theorem.

## 2.2 Uniqueness of Derivation Trees

Recall the following inference rules for typing lambda terms and application for the lambda calculus:

$$\frac{\Gamma, x \mapsto t_1 \vdash e : t_2}{\Gamma \vdash \lambda x. e : t_1 \rightarrow t_2} \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

The second rule introduces a new type above the line.  $t_1$  gets “lost” when building a tree. We might not know what  $t_1$  was used and we don’t have information to know that  $t_1$  was well-typed. For example, there are infinite number of derivations that end up with the term below:

$$\frac{\Gamma \vdash \lambda z. T : (t \rightarrow t) \rightarrow Bool \quad \Gamma \vdash \lambda y. y : (t \rightarrow t)}{\Gamma \vdash (\lambda z. T) (\lambda y. y) : Bool}$$

Mostly, this will be ok. Later, when we introduce subtyping, we will need to pay attention to the number of derivation trees.